Analyzing Software Transactional Memory Applications by Tracing Transactions

Márcio Castro, Kiril Georgiev, Vania Marangonzova-Martin, Jean-François Méhaut, Luiz Gustavo Fernandes, Miguel Santana

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Analyzing Software Transactional Memory Applications by Tracing Transactions

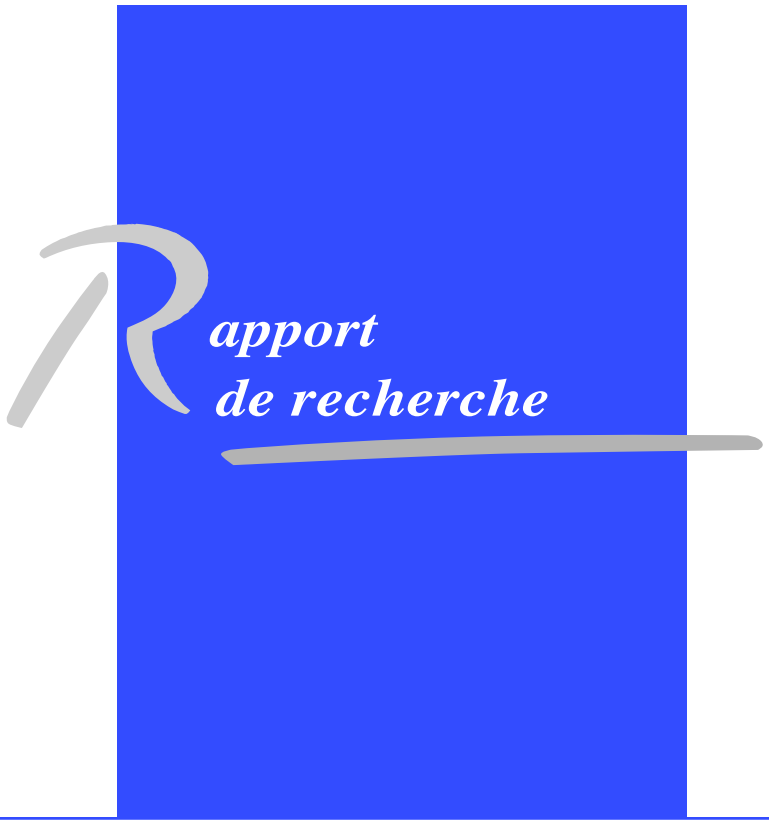
Márcio Castro — Kiril Georgiev — Vania Marangonzova-Martin —

Jean-François Méhaut — Luiz Gustavo Fernandes — Miguel Santana

N° 7334

July 2010

Thème NUM



ISSN 0249-6399 ISRN INRIA/RR--7334--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Analyzing Software Transactional Memory Applications by Tracing Transactions

Márcio Castro*, Kiril Georgiev†, Vania Marangonzova-Martin*,
Jean-François Méhaut*, Luiz Gustavo Fernandes‡, Miguel Santana†

Thème NUM — Systèmes numériques
Équipe-Projet MESCAL

Rapport de recherche n° 7334 — July 2010 — 21 pages

Abstract: Transactional Memory (TM) is a new programming paradigm that offers an alternative to traditional lock-based concurrency mechanisms. It provides a higher-level programming interface and promises to greatly simplify the development of correct concurrent applications on multicore architectures. However, simplicity often comes with an important performance deterioration and, given the variety of TM implementations, it is still a challenge to know what kind of applications can really take advantage of TM. In this work, we aim at investigating these performance issues and presenting a generic approach for tracing transactions. We show that the collected information can be helpful in order to improve the performance of TM applications.

Key-words: Software Transactional Memory, Performance Evaluation, Tracing Mechanism.

* MESCAL (INRIA - LIG - Grenoble University), Grenoble, France, FirstName.LastName@imag.fr
† STMicroelectronics, Crolles, France, Kiril.Georgiev@st.com, Miguel.Santana@st.com
‡ GMAP - PPGCC - PUCRS, Porto Alegre, Brazil, luiz.fernandes@pucrs.br

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Analyse des Applications qui Utilisent la Mémoire Transactionnelle Logicielle par le Traçage de Transactions

Résumé : La Mémoire Transactionnelle (MT) est un nouveau paradigme de programmation concurrente qui vise à être une alternative aux mécanismes traditionnels de synchronisation basés sur des verrous. C'est une approche de plus haut niveau qui permet de simplifier le développement des applications concurrentes sur des architectures multicoeurs. Toutefois, ce haut niveau d'abstraction peut avoir un impact négatif sur la performance et les développeurs de STM font des choix d'impémentation qui ont également un impact important sur la performance. La conséquence est qu'il est difficile de prédire les performances des applications en utilisant la MT. Dans ce travail, nous visons à analyser ces problèmes de performance et nous proposons une approche générique et portable de traçage des transactions. Nous montrons que les informations collectées peuvent être très utiles pour comprendre et améliorer la performance des applications utilisant la MT.

Mots-clés : Mémoire transactionnelle logicielle, évaluation de performance, mécanisme de traçage.

1 Introduction

Multicore technology proves to be a promising solution to the problem of achieving higher performance without increasing power consumption. Because of that, it is strongly expected that the number of processor cores will continue to increase, resulting in manycore architectures with hundreds or even thousands of cores. In this context, the development of applications with high degrees of parallelism and the correct management of complex synchronization issues become a major concern.

Traditional synchronization structures such as *locks*, *mutexes* and *semaphores* are extensively used in a multicore context. They are simple to implement in hardware and they offer a safe solution to the problem of multiple threads sharing data. However, they have several disadvantages: (i) they are "low-level" mechanisms, since one must explicitly control the access to shared variables; (ii) they cause blocking, so threads always have to wait until a lock (or a set of locks) is released; (iii) they are hard to manage effectively, especially in large systems; and (iv) they can be vulnerable to failures and faults, such as deadlocks and livelocks.

Transactional Memory (TM) [1] has recently been proposed as an alternative synchronization solution. The idea is to offer a high level synchronization interface where developers only need to enclose concurrent accesses to shared variables in atomic sections (*transactions*). Problems such as correct synchronization, correct data race handling and deadlocks avoidance are shifted to the TM mechanism, which handles conflicts in an optimistic way [2].

Although TM promises to substantially simplify the development of correct concurrent programs, programmers will still need to debug code and study ways to optimize TM applications. It is clear that even with TM it is still a challenge to design and implement scalable concurrent programs. In this context, the questions we are interested in are the following. How can one know if an application will perform well with TM? How can one get useful details about the execution of TM applications? How can we use them to improve performance?

In this work, we show that the performances of applications using TM-based synchronization solutions depend on both applications and TM solutions specifics. We demonstrate that, depending on these specifics, the use of TM may result in worse, equal or better performance for the application. In order to gain some insight on these issues, helping developers to understand and improve their performance, we propose an approach for collecting and tracing relevant information about transactions. Our solution can be applied to different STM libraries and applications as it does not modify neither the target application nor the STM library source codes.

The rest of this report is organized as follows. In Section 2, we describe the basic idea behind TM along with some important design criteria that impact TM performance. Sections 3 and 4 motivate the use of STM as well as the necessity of tools to better comprehend TM applications. In Section 5, we show our approach for tracing transactions. The collected information and results analysis are shown in Section 6. Section 7 reviews some related works concerning STM. Finally, concluding remarks and future works are pointed out in Section 8.

2 Transactional Memory (TM)

In 1977, Lomet has observed that an abstraction similar to a database transaction might make a good programming language mechanism to ensure the consistency of data

shared among several processes [3]. Sixtheen years after this publication in 1993, Herlih and Moss proposed a hardware-supported Transactional Memory (TM) as a mechanism for building lock-free data structures [4]. In the past few years, there has been a huge interest of researches in implementing both hardware and software systems for Transactional Memory.

This section aims to bring up some important information concerning the concept of Transactional Memory. First, its general concept is presented (Section 2.1). The benefits of using Transactional Memory are discussed in Section 2.2. Finally, the different ways to implement Transactional Memory are shown in Section 2.3.

2.1 What is Transactional Memory?

The basic idea behind Transactional Memory comes from transactional database management systems, in which a transaction is a sequence of actions that appears indivisible and instantaneous to an outside observer. In these systems, two or more queries conflict when different transactions perform read and write instructions over a database in such a way that the result could not arise from a sequential execution of the queries. In this context, transactions ensure that all queries produce the same result as if they executed serially. A database transaction enforces some properties called ACID: atomicity, consistency, isolation and durability.

- *Atomicity*: it refers to the ability of the DBMS to guarantee that either all tasks of a transaction are performed or none of them are performed. It is not acceptable for a constituent action to fail and for the transaction to finish successfully nor it is acceptable for a failed action to leave behind evidence that it executed [2]. Thus, there are two possibilities for an executing transaction: it can be either *committed* (if it completes successfully) or *aborted* (if it fails).

- *Isolation*: it refers to the requirement that other operations cannot access (or see) the data in an intermediate state during a given transaction. Because of that, transactions must produce a correct result, regardless of which other transactions are executing concurrently. This property makes transactions an attractive programming model for parallel computer.

- *Consistency*: this property ensures that the database remains in a consistent state before starting a transaction and after finishing it (whether successful or not). In other words, consistency states that only valid data will be written to the database (integrity constraints). If a transaction has committed, it is guaranteed that the DB had its state modified and this new state is consistent. Thus, subsequent transactions can start executing from that modified state. Otherwise, in case of a transaction abortion, this property is also guaranteed, since the DB did not modify the previous consistent state.

- *Durability*: once a transaction commits, its result must be permanent (*i.e.*, stored on a disk) and available to subsequent transactions. This means that it will survive even if a system failure occurs. Many databases implement durability by writing all transactions into a transaction log that can be played back to recreate the system state right before a system failure.

It is important to mention that *implementations of Transactional Memory do not provide consistency and durability properties* [1]. The durability property does not

make much sense because main memory and caches, which are responsible for storing data during the program execution, are volatile. The consistency property is not usually applied: Transactional Memory does not have a *metadata* concept so there are no independent consistency rules that could be observed during a transaction. If, for example, one wants to implement a transactional queue data structure defining some consistency rules, then the source code needs to take care of enforcing it (it will not be guaranteed by the Transactional Memory).

In the context of TM, a *transaction* is a portion of code that must be executed atomically and with isolation. A transaction may *commit* successfully, if its accesses to shared data did not conflict with other transactions; otherwise the transaction *aborts*, and none of its actions become visible to other threads. When a transaction aborts, the TM runtime *rollbacks* the conflicting transaction until it is possible to commit successfully.

2.2 Why are researchers interested in Transaction Memory?

One of the most serious challenges in writing correct code is to coordinate access to shared data. Depending on the complexity of problem that must be parallelized, more or less mechanisms must be used to guarantee mutual exclusion. As a consequence of that, data races, deadlocks and scalability problems come to light.

Nowadays, the synchronization between threads is the responsibility of a programmer, who has only low-level mechanisms, such as locks, semaphores and mutexes to prevent two concurrent threads from interfering. Some languages have a slightly higher level construct, a monitor, to prevent concurrent accesses. However, these mechanisms are difficult to use correctly and are not composable [2].

The TM programming model offers a new attractive way of developing parallel applications using a higher abstraction level. It shifts the problem of correct synchronisation to the TM system, which is responsible for making sure that deadlocks will not occur, race conditions are correctly handled and locks are performed at a granularity which allows to indeed exploit the inherent parallelism of the application [5].

Since the past few years, there is a growing interest of researchers in improving Transactional Memory. However, researchers still do not fully understand the trade-offs and programming pragmatics of the TM programming model. For instance, the semantics of nested transactions is an area of active debate. Questions such as *"what happens with the outer transaction when a inner transaction aborts?"* and *"if the inner transaction commits, should its results be visible only to the outer transaction or to all transactions?"* motivate interesting debates between researches.

Another issue that is being discussed nowadays is the performance of TM: it is not yet good enough for widespread use. Software TM systems impose considerable overhead costs, since all mechanisms are implemented in software. However, they are hardware-independent, so the same solution can be used in different platforms. On the other hand, hardware TM systems can lower this overhead, but they are only starting to become available, since all mechanisms need specific hardware to implement the concept of Transactional Memory. In this context, hybrid solutions (TM implemented in software that uses specific hardware to reduce the software overheads) appears as a way to balance these issues. The next section will discuss about these different design choices for Transactional Memory.

2.3 Design Choices

Transactional Memory can be implemented in software (*Software Transactional Memory*) [6, 7, 8], in hardware (*Hardware Transactional Memory*) [9, 10] or in both (*Hybrid Transactional Memory*) [11, 12]. Software Transactional Memory (STM) has several advantages over Hardware Transactional Memory (HTM). It offers flexibility in implementing different mechanisms and conflict detection/resolution policies. It is easier to be modified or extended and is not limited by small fixed-size hardware structures, such as cache memories. Finally, STM does not require specific hardware, so it can be used on current platforms.

When designing a TM solution, four important criteria must be taken into account: transaction granularity, version management, conflict detection and conflict resolution.

- *Transaction Granularity*: it defines the unit of storage for conflict detection [2]. For instance, in object-based languages, it is common to use *object granularity*, which detects conflicts when the states of shared objects are modified. Other examples are the *word granularity* and the *block granularity*, which respectively use memory words or groups of words for conflict detection. The transaction granularity cannot only have an important impact on the number of conflicts to be managed but also on the TM overall performance.

- *Version Management*: since a transaction typically modifies data in memory, it is important to control how these modifications are managed on memory. There are two general ways to control it: *eager version management* and *lazy version management*. If the first one is applied, transactions will directly modify data and the system will use some sort of concurrency control to prevent other transactions from concurrently modifying objects. The system records the original data before updating, so it can be restored in case of transaction abortion. If *lazy version management* is used, transactions will deal with private copies of data. When a transaction commits, it updates the original data using the private copy.

- *Conflict Detection*: there are two possible strategies to detect conflicts: *eager conflict detection* and *lazy conflict detection*. The first strategy detects read/write conflicts as they occur whereas the second one only detects at commit time.

- *Conflict Resolution*: after detecting a conflict, the TM system needs to solve it. The usual solution is to abort one or more conflicting transactions. There are many different algorithms that are used to select which transactions must be aborted in order to guarantee forward progress. Usually, a TM system has a specific module called *contention manager*, which implements one or more *contention resolution policies* that are responsible for deciding which conflicting transaction must be aborted. The selected resolution policy clearly affects the performance of a TM system.

To sum up, TM solutions must take care of these issues in order to guarantee a functioning solution. The variety of possible combinations of such criteria clearly affects the behavior of TM applications as well as their performances.

3 STM versus Locks

The performance and benefits of using STM have been discussed since its first proposal in 1993 [13, 1]. In terms of performance, the research community tends to claim that

it always results in considerably higher overheads than locks. However, this statement is not always true and it is not easy to foresee the performance of a TM application.

In this section we consider the well-known *Traveling Salesman Problem* (TSP) [14] in which the goal is to find the shortest possible path visiting each node of a graph exactly once. Here, we aim at comparing the performance of our two different solutions for the TSP: (i) using STM and (ii) using POSIX mutex locks.

In both TSP implementations the graph exploration is done by multiple threads which access shared variables managing the current shortest path and the pool of paths to explore. In the lock-based version, accesses to shared variables are enclosed by Pthread mutex lock/unlock sections. This is the case, for instance, of the accesses to the **minimum** variable, which stores the current shortest path (Listing 1).

Listing 1: Lock-based Accesses.

```
1 void tsp(...) {
2    ...
3    pthread_mutex_lock(&mutex_minimum);
4    if (len < minimum)
5       minimum = len;
6    pthread_mutex_unlock(&mutex_minimum);
7    ...
8 }
```

With STM, accesses to shared variables are enclosed by transactions whose boundaries are indicated by two special functions, *i.e.*, **stm_start()** and **stm_commit()**. During a transaction, shared variables are accessed through the use of two functions: **stm_load()** and **stm_write()**. If we consider again the example of the **minimum** variable, the above lock-based section is transformed in the following STM-based code. It should be noted that locks must be explicitly named by the programmer whereas this is not necessary when using transactions (Listing 2).

Listing 2: STM Access to Shared Data.

```
1 void tsp(...) {
2    ...
3    stm_start();
4    if (len < stm_load(minimum))
5       stm_store(minimum, len);
6    stm_commit();
7    ...
8 }
```

We have carried out several experiments on a Symmetric Multiprocessor machine (SMP) composed of four Intel Xeon X7460 (2.66GHz) processors with six cores each. This platform has 64GB of shared main memory and runs the x86_64 GNU/Linux operating system (kernel 2.6.262). The results were obtained through the average of 30 executions, presenting a low standard deviation.

We have used two approaches in implementing TSP. Our first approach is based on protecting *all* accesses to shared variables with the synchronization mechanisms. Figure 1 shows the execution times we have obtained by running TSP with STM and POSIX mutex locks.

The results show a poor performance of TSP with locks in comparison to STM. This occurs due to the great number of accesses to the shared variable **minimum**, causing

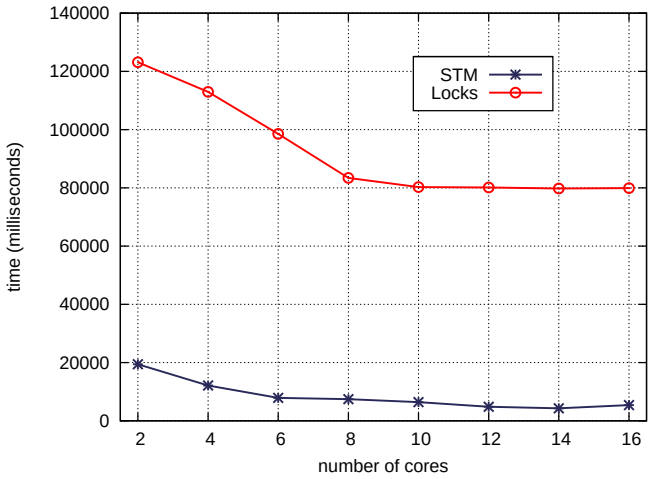


Figure 1: TSP Results: First Approach.

threads to be blocked consecutively. On the other hand, it is not a bottleneck for the STM solution for two reasons: (i) the STM solution uses an optimistic approach to handle multiple accesses to shared data, so it does not block all threads; and (ii) most of the accesses to this shared variable do not conflict, which benefits such optimistic approach.

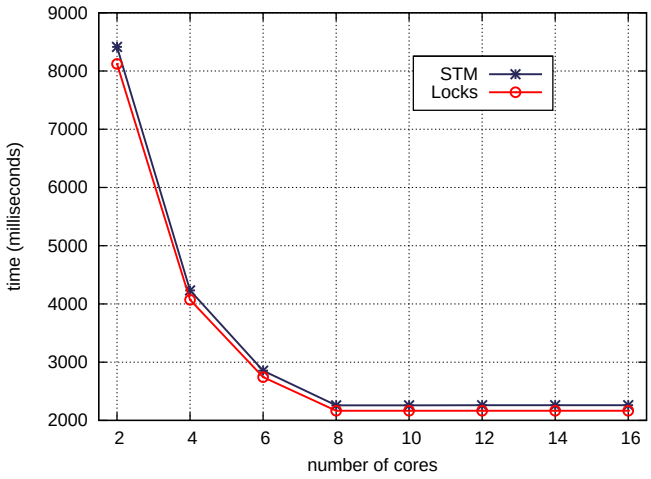


Figure 2: TSP Results: Second Approach.

After a careful analysis, we noticed that the variable **`minimum`** has multiple read-only accesses allowing the removal of some "extra synchronization" without creating data races and then increasing even more the parallelism. This strategy is implemented in our second approach and the execution times are shown in Figure 2.

It can be observed that the performance of the lock-based solution has drastically increased. There are also some slight performance improvements for the STM solution. Both curves are very similar and no considerable overhead has been added by

the STM (it is 5% worse on average), which shows that, depending on the applications characteristics, STM can be as performant as locks.

4 Performance Impact of STM Solutions

In order to investigate the impact of STM solutions on the performance of TM applications, we have carried out experiments with four different non-trivial TM applications available from the *Stanford Transactional Applications for Multi-Processing* (STAMP). The selected applications have been executed with three state-of-art STM libraries, namely TinySTM [8], TL2 [6] and SwissTM [7].

Sections 4.1 and 4.2, give a brief description of each one of the selected STM libraries and applications, respectivelly. Finally, in Section 4.3 we compare the performance of the three STM libraries using four STAMP applications.

4.1 STM Solutions

Usually, STM systems can be either implemented in a library or directly into a compiler. The library-based solution requires the programmer to add explicit calls to the STM library for every access to the memory inside a transaction. This approach is applicable in every system but requires significant changes to the application source code when compared to the sequential code. In order to address this issue, a compiler or preprocessor can be used to automatically convert all accesses to memory that happen within a transaction (*e.g.*, delimited by an *atomic block*) into proper function calls of a STM library. Implementations such as Transactional Locking II (TL2), TinySTM and SwissTM use the library approach.

On the other hand, the second approach is to use a compiler which includes all STM functionalities. In this case, the compiler includes a specific support to STM for a specific programming language. The scope of software support includes language extensions to specify and define transaction regions (*atomic blocks*). The Intel C++ STM Compiler applies this approach.

In the following sections we give a brief description of each one of the STM libraries we have used in our experiments.

4.1.1 Transactional Locking II (TL2)

The TL2 algorithm is a global version-clock based variant of the original transactional locking algorithm of Dice and Shavit (TL) [15]. Based on this global versioning approach, and in contrast with prior local versioning approaches, the authors were able to eliminate several key safety issues afflicting other lock-based STM systems and simplify the process of mechanical code transformation [6].

The basic idea of TL2 algorithm is to use a *global version-clock* counter in order to handle conflicts between transactions. The counter is incremented using an *increment-and-fetch* function implemented with a *compare-and-swap* (CAS) operation. This global variable will be read and incremented by each *writing transaction* and will just be read by every read-only transaction.

A similar idea of the global version-clock variable is used to implement a data structure responsible for storing a collection of special *versioned write-locks* used for every transacted memory location. In this case, a single bit is used to indicate whether the lock is taken. The rest of the lock word is used to store a version number. This

version number is incremented by every successful release of the respective lock. It is also possible to associate locks and shared data in different ways such as *per object* (a lock is assigned per shared object) or *per stripe* (the shared memory is partitioned using some hash function to map a striped location to a lock).

The sequence of operations that are performed by the TL2 when a transaction begins depends on the type of the transaction. One of the goals of the TL2 algorithm is to offer an efficient execution of *read-only transactions*. In this case, few steps are executed allowing low-cost read-only transactions. On the other hand, if it is a *write transaction*, that means a transaction that performs writes to the shared memory, more steps are needed and it may have a significant cost depending on the operations that are executed inside the transaction.

4.1.2 TinySTM

TinySTM is another well-known STM implementation that also uses a global versioning approach (shared counter as clock) to control the conflicts between transactions and locks to protect shared memory locations [8]. Since the maximal value of the clock is 2^{31} on a 32-bit architecture and 2^{63} on a 64-bit architecture, it may be quickly reached in 32-bit systems with frequent commits. TinySTM avoids this problem by implementing a mechanism to automatically reset the clock when the maximal value is reached.

Two strategies for accesses to memory have been implemented in TinySTM: *write-through* and *write-back*. The first strategy allows transactions to directly write to memory and revert their updates in case they need to abort whereas the second delays memory updates until commit time. It is also important to mention that each strategy has its advantages and limitations. Write-through has lower commit-time overhead, faster read-after-write/write-after-write handling and enables many interesting compiler optimizations. On the other hand, write-back has lower abort overhead and does not require extra techniques to guarantee consistent reads.

As TL2, TinySTM uses a shared array of locks to manage concurrent accesses to memory. Each lock covers a portion of the address space and the addresses are mapped to locks based on a hash function. Since write-transactions must verify that all the addresses they have read are still valid (*i.e.*, they are not locked by another transaction and still have the same version number) at commit time, depending on the number of read and write operations, this verification may be costly. In order to address this issue, the authors propose a hierarchical locking strategy. In this strategy, leaves of the tree (last level on the hierarchy) correspond to elements of the shared array of locks while upper levels aggregate information about lower levels. Thus, when there is no lock acquired for any element of a given sub-tree, there is not necessary to validate its elements.

In [8], the authors also emphasize the importance of tuning some TinySTM parameters and show that it can really impact on the transaction throughput. The three most important parameters are: the hash function which maps a memory location to a lock, the number of entries in the lock array and the size of the array used for the hierarchical locking. Because of that, the authors propose a dynamic tuning strategy to automatically adjust these parameters according to the workload and show that it results in important performance gains.

4.1.3 SwissTM

SwissTM [7] is a very recent STM implementation which has some similar characteristics when compared to TL2 and TinySTM. It is a lock-based STM, which means that it uses a lock table to manage concurrent accesses to memory. As TL2 and TinySTM, the API of SwissTM is also word-based, as it enables transactional access to arbitrary memory words (word granularity).

However, SwissTM presents some new features when compared to other libraries such as TL2 and TinySTM. One of its innovations is the hybrid conflict detection scheme: it detects write/write conflicts eagerly, which prevents transactions that will probably abort from running and wasting resources, and read/write conflicts lazily, allowing more parallelism between transactions. In read/write conflicts, a time-based scheme (similar to the TL2 global version-clock) is applied to handle conflicts.

Another distinctive feature of SwissTM is its two-phase contention manager. In brief, short or read-only transactions use the simple but inexpensive *timid contention management* scheme, aborting transactions when the first conflict is encountered. On the other hand, more complex transactions are switched dynamically to a mechanism that involves more overhead but favors theses transactions, preventing starvation. Specifically, the mechanism that is applied for such complex transactions is called Greedy [16].

It is also important to mention that SwissTM has the *weak atomicity* property. This means that it cannot detect conflicts between accesses to memory inside and outside transactions. In order to guarantee all properties of transactional memory, it is necessary to perform all accesses to shared variables inside transactions.

4.2 Transactional Memory Applications

STAMP is a benchmark which includes 8 applications developed for TM [17]. It offers several advantages in comparison with other benchmarks: (i) the applications use a variety of algorithms and belong to different application domains; (ii) it is possible to simulate different transactional behaviors varying the size of transactions (in terms of the number of instructions), the amount of contention and their granularity; and (iii) the applications can be easily executed with different STM libraries. In this work, we have selected the following STAMP applications:

- *genome:* it takes a large number of DNA segments as its input parameter and tries to mach them to reconstruct the original source genome. This process is composed by two phases. The first phase of the algorithm uses a hash set to create a set of unique segments, excluding all duplicates. After that, the second phase is executed by many threads where each one tries to remove a segment from a global pool of unmatched segments and add it to its partition of currently matched segments. In order to match segments fastly, the algorithm uses Rabin-Karp string matching. Additions to the set of unique segment and accesses to the global pool of unmatched segments are enclosed by transactions to allow concurrent accesses. This application is characterized by medium-sized transactions and it spends most of its execution time executing transactions.

- *intruder:* it emulates Design 5 of the Signature-based network intrusion detection system, which scans network packets in order to detect a known set of intrusion signatures. It is composed by three phases: capture, reassembly and detection. Different shared data structures are used depending on the phase: a FIFO

queue is used in capture whereas a dictionary implemented by a self-balancing tree is used in reassembly phase. Both capture and reassembly phases are enclosed by transactions. This application is composed by considerably fewer (but bigger) transactions than *genome*.

- *labyrinth:* it is a variant of Lee's routing algorithm [18] implemented with transactions. The calculation of the path is enclosed by a single transaction and a conflict occurs when two or more threads pick paths that overlap. Transactions are beneficial for implementing this solution since deadlock avoidance techniques are required when implementing it with locks. It has very different characteristics: short transactions (few instructions inside transactions) but it is composed by an extremely large number of transactions in comparison to the others.

- *ssca2:* it is composed by four graph kernels that operate on a large, directed, weighted multi-graph. In this report, we have selected the Kernel 1, which is responsible of constructing an efficient graph data structure using adjacency arrays and auxiliary arrays. In STAMP, the parallel transactional version of such kernel is composed by threads that add nodes to the graph in parallel. In this context, transactions are used to protect accesses to the adjacency arrays.

These applications have been selected since they present different characteristics in terms of computation, transactions lengths (**Tx Length**) and time spent in transactions (**Tx Time**). A summary of these characteristics is shown in Table 1.

Table 1: Applications characteristics.

Application	Tx Length	Tx Time	Input parameters
genome	Medium	High	genome -g256 -s16 -n16384
intruder	Short	Medium	intruder -a10 -l4 -n2048 -s1
labyrinth	Long	High	labyrinth -i random-x512-y512-z7-n512.txt
ssca2	Short	Low	ssca2 -s14 -i1.0 -u1.0 -l9 -p9

A complete description of each application as well as the possible input parameters and their influences on the performance can be found in [17].

4.3 Performance Analysis

In this section we show the results obtained by carrying out experiments with the four applications and three STM libraries. For this purpose, we have used the same SMP machine described in the previous section and the results were also obtained through the average of 30 executions (an insignificant standard deviation has also been observed).

Figure 3 (a) shows the speed-ups we have obtained by executing *genome* with the three STM libraries. For this particular application, TinySTM has presented the best speed-up (9.75 with 16 cores) and better scalability than the others. We can notice that TinySTM and SwissTM perform still better with 16 cores, while TL2, on the contrary, presents a considerable performance degradation.

The results of the *labyrinth* executions are shown in Figure 3 (b). Unlike the previous results, SwissTM presents the best performance gains (7.71 with 16 cores). TL2

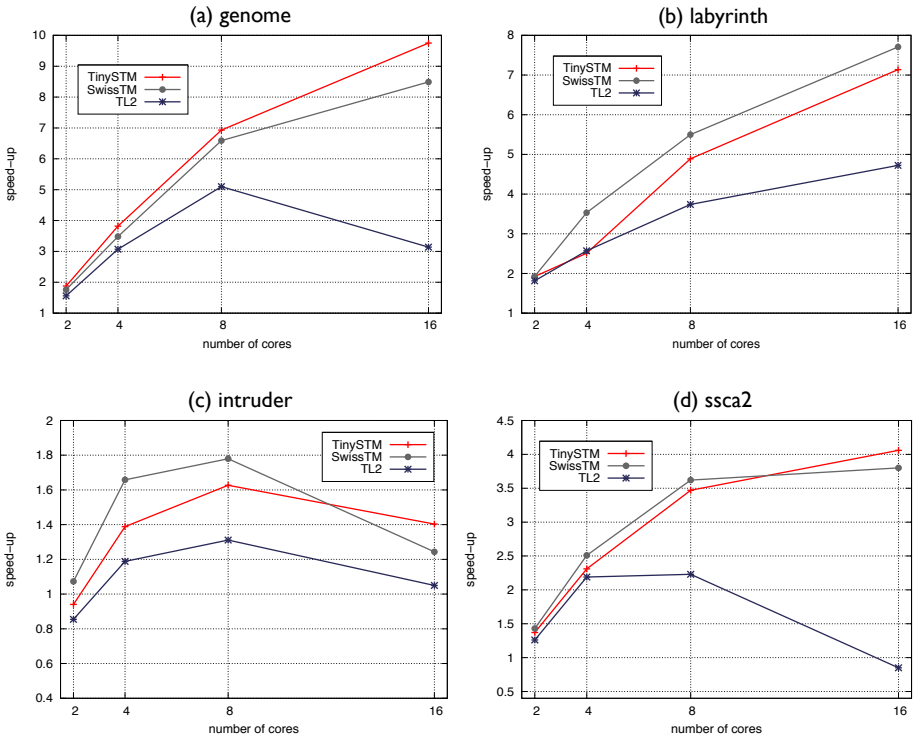


Figure 3: Performance of the Selected STAMP Applications.

has shown better scalability in comparison to the *genome* results but it has still presented poor performance compared to the other STM libraries.

In Figure 3 (c), we present the speed-ups obtained while executing *intruder*. We can observe a very different behavior: there are almost no performance gains as the speed-ups obtained do not exceed 1.8. SwissTM has presented better performance gains in comparison to TinySTM from 2 to 8 cores. However, TinySTM has surpassed the performance of SwissTM when using 16 cores.

Finally, in Figure 3 (d), we present the results obtained by running *ssca2*. As in *intruder*, SwissTM has presented better speed-ups with few processor cores. A maximum speed-up of 4.1 has been achieved with 16 cores. Like *intruder*, this application also presented poor scalability which indicates that it does not take advantage of the STM optimistic approach.

Regarding the results presented above we can conclude that it is not trivial to predict the performance of a TM application. The characteristics and design choices of the STM library can undoubtedly change its performance. In this context, we argue that such characteristics must be taken into account in order to develop a performant TM application. Thus, it emerges the necessity of having ways to better comprehend their behavior. In order to obtain some insight on such issues, we propose an approach for collecting and tracing relevant information about transactions.

5 Tracing Transactions

Tracing applications basically consists in recording a chronological history of events, representing the application behavior. An event is an action during the execution of an application that changes its state. In this work, we are specifically interested in events deriving from the use of STM. In the following sections, we describe how these events can be traced. Firstly, we specify the desired characteristics of our approach. Secondly, we explain the general functioning of STM libraries and what events we intend to trace. Finally, we describe our tracing mechanism.

5.1 Goals

Our mechanism aims to tackle two relevant issues:

- *STM Application and STM Library Independency:* we want a tracing solution which does not change neither the TM application nor the STM library source codes. In order to do so, we have chosen to implement an interception mechanism placed between the STM application and the STM library.
- *Intrusiveness:* our tracing solution should minimize intrusiveness meaning that it should not imply an important execution overhead. Indeed, when that overhead is important, the application behaves differently and the traces may not represent the real application behavior. In order to minimize intrusiveness, we have decided trace a very reduced set of events.

5.2 Traced Events

We have selected the two most important functions to be traced, *i.e.*, **`stm_start()`** and **`stm_commit()`**, which respectively indicate the beginning and the end of transactions. We believe that, by intercepting them, relevant information about TM applications can be extracted without increasing the degree of intrusiveness. In the following paragraphs we explain more in detail the functioning of these two functions.

The function **`stm_start()`** is responsible for initializing the transaction specific structures, as well as for saving the calling environment for later use in case of abort. It typically saves the stack context containing the current thread local program counter and registers values. The role of the **`stm_commit()`** function is to verify whether the current transaction is in conflict with any other transaction.

When a conflict occurs, a rollback mechanism calls **`stm_start()`** in order to restart the transaction. The system rollbacks the transaction by restoring the environment that has been saved by the first call to **`stm_start()`**. In order to do that, **`stm_start()`** and **`stm_commit()`** use the Linux system calls **`sigsetjmp()`** and **`siglongjmp()`** to respectively save and restore the thread environment. This strategy is applied in the majority of STM libraries, including TinySTM, TL2 and SwissTM. When there is no conflicts, all changes that have been done by the transaction are made permanent (*validation*).

We can easily obtain the number of commits and aborts during the application execution, since aborts can be identified by successive calls to **`stm_start()`** in the context of a specific thread. With such information, we can deduce other metrics such as the rate of successfully committed transactions, the time spent re-executing transactions, etc.

5.3 Tracing Mechanism

Our tracing solution is based on the Linux dynamic linking mechanism which provides a simple way to intercept function calls. It provides the environment variable called **`LD_PRELOAD`** which is used to dynamically load a library LIB when launching applications. During the execution, the system will intercept the functions having the same signatures as the ones implemented in LIB, calling the corresponding LIB functions (*wrappers*). Wrapper functions may implement their proper behavior but it is still possible to call the original ones.

In our case, a shared library called **`libTraceSTM.so`** has been implemented, containing two wrappers for the **`stm_start()`** and **`stm_commit()`** functions. By executing **`LD_PRELOAD=./libTraceSTM.so app`** (where **`app`** is the target STM application), the original STM functions are dynamically overridden by our wrapper functions. The wrappers are responsible for tracing and calling the corresponding original functions, *i.e.*, **`stm_start()`** and **`stm_commit()`**.

Listing 3: STM Function Wrappers.

```
1  void stm_start() {
2     realStmStart = dlsym(handle, "stm_start");
3     ...
4     pthread_mutex_lock(&trace_lock);
5     trace("stm_start");
6     (*realStmStart)(); //calls the real function
7     pthread_mutex_unlock(&trace_lock);
8  }
9
10 void stm_commit() {
11    realStmCommit = dlsym(handle, "stm_commit");
12
13    pthread_mutex_lock(&trace_lock);
14    ...
15    (*realStmCommit)(); //calls the real function
16    ...
17    trace("stm_commit");
18    pthread_mutex_unlock(&trace_lock);
19 }
```

Listing 3 shows the two selected wrapper functions: **`stm_start()`** (line 1) and **`stm_commit()`** (line 10). In **`stm_start()`**, we first obtain a handle to the original STM function (line 2), trace the related event using the **`trace()`** function (line 5) and call the original STM function (line 6). In **`stm_commit()`**, however, we first call the original STM function before tracing it. As explained before, **`stm_commit()`** rollbacks in case of conflicts and **`stm_start()`** is called again. So, tracing events after **`stm_commit()`** ensures that transactions have already committed successfully. Successive calls to **`stm_start()`** in the context of the same thread indicate transactions that have been aborted.

It is important to notice that calls to **`trace()`** and the real STM functions must be **atomic**. Otherwise, the order of recorded events may not correspond to the real sequence of calls to STM functions. This is the reason why we use locks and some additional treatments in order to avoid deadlock situations that can easily arise during rollbacks.

Figure 4 shows our tracing mechanism. When a thread is initialized by the STM application, our shared library adds the thread ID in an internal data structure, creates a trace file and instantiates a circular memory buffer. When subsequent STM function

calls are intercepted, the trace record is written into the corresponding thread's circular buffer. When the circular buffer is full, its contents are flushed to the corresponding trace file.

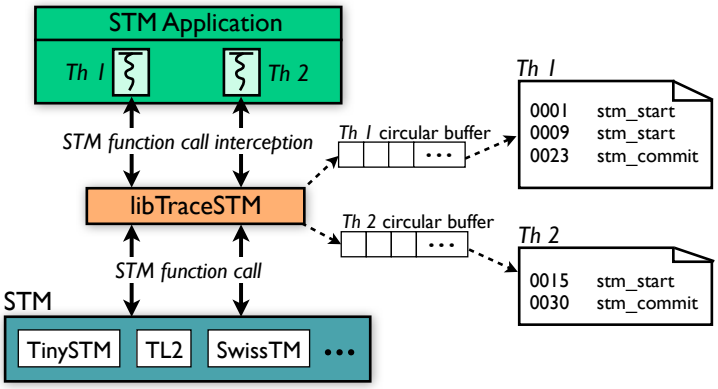


Figure 4: Overview of the Tracing Mechanism.

Each event to be traced is represented by a timestamp and the name of the intercepted function. As we target shared memory multithreaded applications, timestamps correspond to the value of the machine's clock. At the end of the execution, we obtain a set of files, one per thread. This is illustrated in Figure 4, which shows a STM application with 2 threads (named $Th1$ and $Th2$). The trace file of $Th1$, for example, shows two successive calls to **`stm_start()`** followed by a **`stm_commit()`**, indicating a transaction that has been aborted once. The trace file of $Th2$ shows a transaction that has been started and committed successfully.

After the execution of the application, we merge the individual trace files, sorting events by their timestamps. In the merged trace file, each event is represented by a timestamp, a thread ID and the name of the intercepted function. Figure 5 shows the final trace that has been derived from the merge of the individual trace files of $Th1$ and $Th2$.

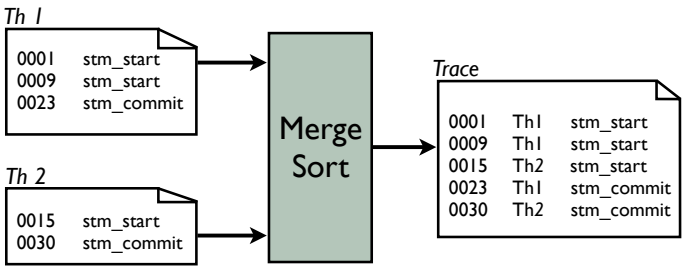


Figure 5: Final Trace Generation.

We believe that our approach to intercept STM function calls is very simple and it can be easily extended if more functions should be traced. It is also generic enough, since it can be used with all STM libraries (*e.g.*, TinySTM, TL2 and SwissTM) and it does not change neither the STM application, nor the STM library source codes.

6 Experimental Results

This section presents the experimental results we have obtained by using our tracing mechanism with the STAMP applications described in Section 4. For this purpose, we have used TinySTM, executing all applications with 16 threads.

Our tracing mechanism allows us to obtain different metrics and statistics about the execution of STM applications. For instance, we can calculate the number of transactions or the number of commits and aborts. We can also observe the wasted work, *i.e.*, the percentage of the transactions execution time that has been spent executing aborted transactions (total and per thread). Other accessible metrics concern the evolution of the number of aborts and commits and the instantaneous commit rate (the proportion of committed transactions at sample points) during the execution.

In this paper, we present the evolution of the number of commits and aborts during the execution. We believe that such information can be very helpful since the number of aborts is one of the most important metrics that influences the performance of STM applications. In the following, we show the results corresponding to the three STAMP applications.

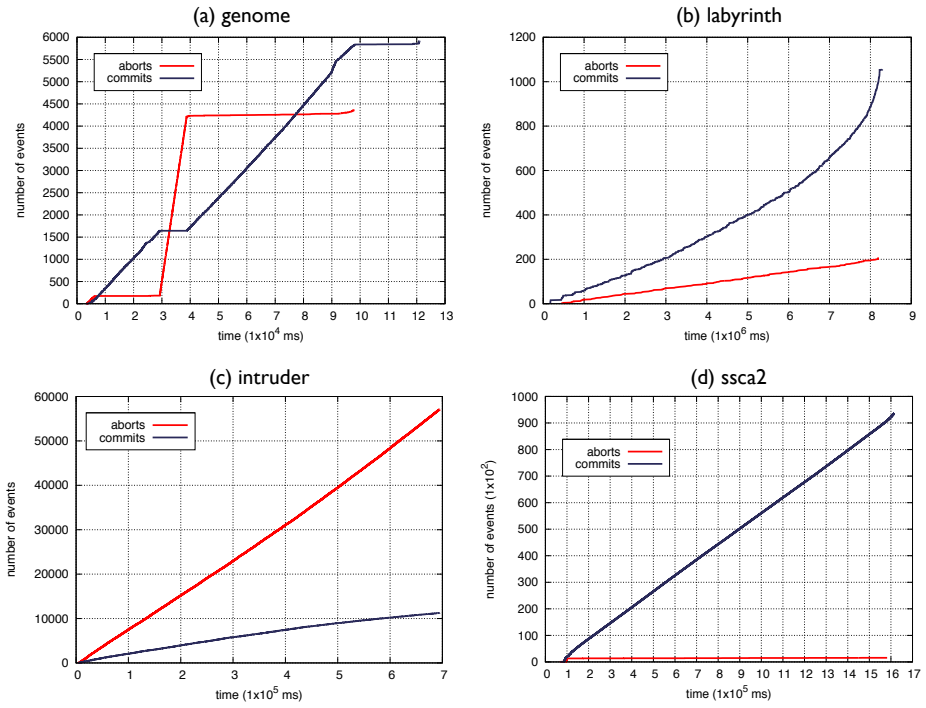


Figure 6: Trace Results: Aborts and Commits Evolution.

Figure 6 (a) concerns the *genome* application. What may be observed is that the commit/abort behavior changes during the execution. Namely, the number of aborts increases drastically between 3×10^4 *ms* and 4×10^4 *ms*. This suggests that, during this period, the probability of having conflicting transactions is very high. However, since the time period is short compared to the total execution time, it does not interfere considerably with its overall performance (as seen in Section 4).

The results obtained with *labyrinth* are shown in Figure 6 (b). As it can be seen, this application takes advantage of the optimistic approach of TM, since the aborts curve is always placed under the commits curve. The difference increases towards the end of the execution (exponential growth). The few number of aborts in comparison to the commits justifies the performance obtained in Section 4.

Traces obtained from *intruder* are shown in Figure 6 (c). Its poor performance observed in Section 4 is confirmed by the presence of a very high number of aborts in comparison to commits, which means that it does not take advantage of the TM optimistic approach.

Finally, the *ssca2* results are shown in Figure 6 (d). Considering the same intuition, we would expect an important number of aborts, since the performance of such application was not satisfactory (Section 4). However, the obtained traces show a very low number of aborts in comparison to commits and it does not reflect its poor performance gains. We can conclude that its poor performance has not been originated by the STM (no overhead has been added) and similar performance gains would be obtained if *ssca2* was implemented with locks.

7 Related Work

Considering that TM is an emerging research area, in the majority of cases, works have been based on proposals of different TM solutions and algorithms. Recently, some works have addressed the performance analysis of different TM solutions and/or TM applications, as in [17], [19] and [20]. However, few works have been done concerning tools to help the development using TM. That is the case of the proposals presented in [21] and [22].

Minh *et al.* [17] have described the eight non-trivial STAMP applications, showing their performance gains with different TM systems and configurations. However, they have used a multicore simulator for all experiments instead of a real multicore platform.

Chung *et al.* [19] have studied 35 benchmarks from different domains. In that work, the authors have translated the original synchronization mechanisms applied on all benchmarks to TM. However, they have neither studied the non-trivial TM applications presented here nor they have analyzed aborts and commits, which are very important metrics.

Marathe *et al.* [20], on the other hand, have compared the performance of their STM solution with other STM implementations on two multicore machines. The performance analysis has been based on four simple micro-benchmarks, which do not represent the behavior of real applications though.

Our work differs from these three, since we have tackled both issues: a performance analysis of four non-trivial TM applications by using three different state-of-art STM libraries over a real multicore machine.

Concerning the study of the behavior of TM applications, we can highlight two very recent papers. On both works, the authors have proposed solutions to profile the execution of STM applications.

Ansari *et al.* [21] have manually instrumented the DSTM2 STM library to collect relevant information during the execution of the applications. They have chosen three applications from STAMP and an implementation of Lee's routing algorithm, investigating some relevant metrics to comprehend TM applications.

Lourenço *et al.* [22] have proposed a monitoring framework, which collects the transactional events into a log file as well as a tool to visualize the results. Their instru-

mentation mechanism is based on a API, so the user must insert the tracing function calls within applications source codes.

Unlike these two proposals, our solution uses an interception approach based on the Linux dynamic linking mechanism. By using such method, we can achieve the STM application and STM library independency, since it does not change neither the TM application nor the STM library source codes and it can be easily applied with different STM libraries.

8 Conclusion and Perspectives

In this paper, we have shown that the performance of applications based on STM is related to two issues: the application itself and the STM library. A TM application that takes into account the characteristics of the underlying TM system may benefit of its optimistic approach, reducing the probability of having conflicts and then, resulting in better performance. On the other hand, we have seen that an application may also behave differently depending on the STM library, which means that the developer must be aware of how the STM library works to achieve the desirable performance.

In order to obtain a better understanding of the performance of STM applications, we have proposed an approach for collecting relevant information about transactions. It is based on a shared library which is dynamically linked with the STM application. Events to be traced are implemented as wrapper functions, which can be easily extended if other events must be traced. Moreover, our solution can be applied to different STM libraries and applications as it does not modify neither the target application nor the STM library source codes.

The selected events (`stm_start()` and `stm_commit()` function calls) allow the extraction of different details and statistics about the execution of a STM application. Specifically, in this work we were interested in analyzing the behavior of aborts and commits during the execution, since they represent an important role on the performance of STM applications.

The collected information representing each event allows a general comprehension about the behavior of all transactions. However, we cannot correlate each event to its corresponding transaction in the context of a specific thread, since transactions are not explicitly identified. As a future work, we intend to study ways of discerning transactions in our tracing mechanism, so finer information can be obtained. Moreover, we aim at investigating the behavior of other non-trivial TM applications, proposing general guidelines to reduce conflicts by analyzing TM applications. Finally, we also plan to study what support we would need in order to use our trace mechanism with hardware and hybrid TM solutions.

References

[1] J. Larus and C. Kozyrakis, "Transactional Memory: Is TM the Answer for Improving Parallel Programming?" *Communications of ACM*, vol. 51, no. 7, pp. 80–88, 2008.

[2] J. Larus and R. Rajwar, *Transactional Memory (Synthesis Lectures on Computer Architecture)*, 1st ed. Madison, USA: Morgan & Claypool Publishers, 2007.

[3] D. B. Lomet, "Process structuring, synchronization, and recovery using atomic actions," *SIGPLAN Not.*, vol. 12, no. 3, pp. 128–137, 1977.

[4] M. Herlihy, J. E. B. Moss, J. Eliot, and B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993, pp. 289–300.

[5] Q. Meunier and F. Pétrot, "Lightweight Transactional Memory Systems for Large Scale Shared Memory MPSoCs," in *To appear: Proceedings of the IEEE NEWCAS-TAISA '09*, Toulouse, France, 2009.

[6] O. S. D. Dice and N. Shavit, "Transactional Locking II," in *DISC '06: Proc. of the 20th International Symposium on Distributed Computing*, 2006, pp. 194–208.

[7] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching Transactional Memory," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 155–165, 2009.

[8] P. Felber, C. Fetzer, and T. Riegel, "Dynamic Performance Tuning of Word-Based Software Transactional Memory," in *PPoPP '08: Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 237–246.

[9] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun, "Characterization of TCC on Chip-Multiprocessors," in *PACT '05: Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[10] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: Log-based Transactional Memory," in *HPCA '06: Proc. of the 12th International Symposium on High-Performance Computer Architecture*, 2006, pp. 254–265.

[11] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid Transactional Memory," in *PPoPP '06: Proc. of Symposium on Principles and Practice of Parallel Programming*, 2006.

[12] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. S. III, and M. F. Spear, "Hardware Acceleration of Software Transactional Memory," in *TRANSACT '06: Proc. of the 1st ACM SIGPLAN Workshop on Languages, Compiler and Hardware Support for Transactional Computing*, 2006.

[13] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software Transactional Memory: Why Is It Only a Research Toy?" *Queue*, vol. 6, no. 5, pp. 46–58, 2008.

[14] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton, NJ, USA: Princeton University Press, 2007.

[15] D. Dice and N. Shavit, "What really makes transactions faster?" in *Proceedings of the 1st TRANSACT 2006 Workshop*, 2006.

[16] R. Guerraoui, M. Herlihy, and B. Pochon, "Toward a theory of transactional contention managers," in *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2005, pp. 258–264.

[17] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *IISWC '08: Proc. of The IEEE International Symposium on Workload Characterization*, 2008.

[18] X. Ji-Guang and T. Kozawa, "An Algorithm for Searching Shortest Path by Propagating Wave Fronts in Four Quadrants," in *DAC '81: Proc. of the 18th Design Automation Conference*. Piscataway, NJ, USA: IEEE Press, 1981, pp. 29–36.

[19] J. Chung, H. Chafi, C. C. Minh, A. Mcdonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun, "The Common Case Transactional Behavior of Multithreaded Programs," in *HPCA '06: Proc. of the 12th International Conference on High-Performance Computer Architecture*. IEEE Computer Society, 2006.

[20] V. J. Marathe and M. Moir, "Toward High Performance Nonblocking Software Transactional Memory," in *PPoPP '08: Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2008, pp. 227–236.

[21] M. Ansari, K. Jarvis, C. Kotselidis, M. Luján, C. Kirkham, and I. Watson, "Profiling Transactional Memory Applications," in *PDP '09: Proc. of the 17th International Conference on Parallel, Distributed, and Network-based Processing*, 2009, pp. 11–20.

[22] J. Lourenço, R. Dias, and J. Luís, "Understanding the Behavior of Transactional Memory Applications," in *PADTAD '09: Proc. of the 2009 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, 2009.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399